ELSEVIER

Parallel Computing 21 (1995) 1451–1483

PARALLEL COMPUTING

A load balanced multicomputer relational database system for highly skewed data *

Fotis Barlos [a], Ophir Frieder [b,*]

[a] *Thinking Machines Corporation, Cambridge, MA, USA*
[b] *Computer Science Department, George Mason University, Fairfax, VA, USA*

Received 9 June 1993; revised 24 May 1994, 29 November 1994

Abstract

Query optimizers are used in most modern high performance databases. The emergence of multiprocessor machines allows the development of more aggressive database designs, which in turn need very complicated query optimizers. We present the Workload Partitioning Segment of DOME, a query optimization environment developed on an Intel i860 hypercube system. DOME calculates the distribution of the query input relations and uniformly allocates the workload associated with the resolution of a query on the nodes of a multiprocessor system. The workload is uniformly allocated even when the input relations exhibit a high degree of skew. A priori information of the degree of skew is unnecessary. DOME provides routines to perform the relational operations in parallel on all nodes of the hypercube system. We tested DOME with various sizes, and degree of skew of the input relations. We describe the theoretical foundation behind the optimization techniques employed and present graphs that portray the performance of DOME and its scalability.

Keywords: Database system; Query optimizer; Workload balancing; Distributed memory system; Intel i860 hypercube

1. Introduction

In this paper we examine a dynamic load balancing optimizer for parallel/distributed database systems. Databases handle data in an efficient way to provide the user with the information requested. User requests are called queries. Opti-

* This work was partially supported by the National Science Foundation under contract number CCR9109804.

* Corresponding author. Email: ophir@cs.gmu.edu

SSDI 0167-8191(95)00028-3

mizers are software systems that determine the most efficient implementation of a query.

Database systems have three levels of abstraction between the storage and handling of raw bits and bytes and the intelligent user interface. The lower level is called physical level, and it contains all the disk file handling routines. The next level is called the conceptual level and manages the database requests based on the model used for the implementation of the particular application. Finally, the higher level, the view level, is the interface with the user. Our research focuses on the conceptual level. Various models have been proposed for the operations of the conceptual level [10,13,18,31]. We focus on the *Relational* model [10], because it has a very solid mathematical formulation, it is widely used, and can deliver ad hoc queries.

In the relational model [10], a relation $\Re$ on the set $\Im$ is a subset of the Cartesian product of $\mathrm{dom}(A_0) \times \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)$, where $\mathrm{dom}(A_i)$ is the domain of A_i. $R[A_0\ A_1\ A_2 \ldots A_n]$ represents $\Re$ on the set $\{A_0, A_1, A_2, \ldots, A_n\}$, and is referred to as a schema of $\Re$. In $R[A_0\ A_1\ A_2 \ldots A_n]$, each column A_i is called attribute of R, and is denoted by $R.A_i$. Each row of R, namely a tuple, is designated by $\langle a_0, a_1, \ldots, a_n \rangle$, where $a_i \in \mathrm{dom}(A_i)$. The value of attribute A_i of tuple $x \in R$ is denoted as $x[A_i]$.

Three of the more common operators in the relational model include the Select, Project, and Join. These three operators are formally defined as follows:

- *Select*: The selection on a relation R[XYZ], denoted by $\sigma_{A=a}(R)$, is defined by

$$\sigma_{A=a}(R) = \{x \mid x[A] = a, x \in R\},$$

where A is an attribute of R.

- *Project*: The projection on R[XYZ], denoted as $\pi_A(R)$, is defined by

$$\pi_A(R) = \{x[A] \mid x \in R\},$$

where A is a set of attributes of R.

- *Join*: The join of two relations R[XYZ] and S[VWX], denoted as R[XYZ] $\bowtie$ S[VWX], is defined by

$$R[XYZ] \bowtie S[VWX] = \{x \mid x[VWX] \in S \text{ and } x[XYZ] \in R\},$$

where V, W, X, Y, and Z are a disjoint set of attributes. If no common joining attributes exists, the Join of R and S is the Cartesian product of R and S.

The need for query optimizers is eminent today. Highly voluminous databases, on the order of tens of terabytes of information, are likely to be common place in the near future. Usually, complex queries can have many implementation strategies, some of them might require the manipulation of a tremendous volume of data, while others much less processing.

The most common optimizers usually assume that the data of the input relations follow a uniform distribution. That is, that the probability of finding a particular record is the same for any record in the relation under search. In practice, this is not always the case. There are situations where the data are heavily skewed. In such cases, the probabilities associated with different records vary significantly.

Examples include bibliographic databases indexed by author name and telephone directories. In such databases, citations under names like "Smith" will probably outnumber entries under names like "Kerby".

Next generation database applications often require the execution of complex queries that process several relations with huge amounts of information. Queries are expressed in the form of graphs. These graphs display the operations that need to be performed among the relations that participate in the query. Implementation of a query is the transformation of such a graph into a tree structure that determines the sequence of operations need to be executed.

In a multiprocessor environment, the operations are decomposed into numerous tasks that are executed by the processors of the system. An efficient query optimizer for such a multiprocessor system must therefore coordinate the execution of these tasks, and it usually confronts the following three challenges:

- Identify an efficient partitioning of the whole workload into equal tasks that are distributed to the nodes of the multiprocessor system and achieve a balanced allocation.
- Identify the query execution plan that requires the least processing resources.
- Identify an allocation scheme of tasks to nodes that requires the minimum communication transactions.

DOME addresses the above three issues. We implemented and tested DOME on a shared nothing environment [30] with exclusive disk storage at each node. Although DOME can operate on any multiprocessor system, we used the Intel i860 hypercube systems as the implementation platforms. The i860 system consists of 32 nodes, each equipped with an Intel i860 processor and 8 Mbytes of main memory. The i860 system does not have local file systems on the nodes. To emulate local file systems we developed a ramdisk library. The ramdisk library provides a transparent disk environment that is required for any real database operations. Through the use of the ramdisk library, all the relations of the database are maintained in the main memory of the nodes.

In our experimental environment, the database relations are horizontally partitioned and distributed on the local file systems of the hypercube. The initial partitioning and distribution is random. DOME must determine an efficient partitioning and distribution schema for the current relational operation, perform the partitioning and distribution based on the derive schema and lastly execute the operation.

The series of operations performed by DOME during the service of a query are outlined below:

- *Preprocessing phase*: A user queries the database. The Database Manager, a software environment acting as a pre-processor, receives the query. The Database Manager translates the query into a series of Query Execution Trees (QET). The services of the preprocessing include other optimizers that narrow down the number of possible QETs. The various QETs are transmitted to the nodes of the hypercube system for manipulation by DOME. Meanwhile, the Database Manager performs a random sampling of the participating relations and provides the resulting sampled relation to DOME.

- *Workload Partitioning phase*: DOME uses the samples obtained during the preprocessing phase to identify, for each QET, a workload partitioning that achieves an even allocation of jobs to the processors.
- *Operation Ordering phase*: DOME uses the partition ranges to execute in parallel every QET on the sampled relations to identify the one with the minimum processing time, named Minimum Query Execution Tree (MQET). Since the sample sizes are minimal, as compared to the size of the full relations, the time required to perform this operation is small as compared to the execution time of the QET with the full relations.
- *Site Selection phase*: DOME allocates the partial workloads of the MQET to the appropriate nodes of the hypercube system for processing. To achieve an efficient allocation, DOME identifies and assigns each partition to the node that maintains the bulk of the data. This allocation strategy is executed for each relational operation of the MQET.
- *Execution phase*: Every relational operation of the QET is executed in parallel by the nodes of the hypercube system. Each node operates only on the data within its partition.

DOME consists of three major segments that perform the above operations:

- *Workload Partitioning Segment* (*PART*), performs the workload partition of the relational operations. During the workload partition phase the sampled data are transferred from the disks to the nodes. Moreover, the nodes communicate to broadcast the relations along the hypercube system. Since the sampled relations are small, the node intercommunication activity and the space requirements for the storage of the sample relations in low.
- *Operation Ordering Segment* (*OPRDER*), identifies the QET with the minimum-most.
- *Site Selection Segment* (*SITESEL*), allocates the jobs of the MQET to processors. During the site selection phase, partitions of the full relations are transferred among the nodes. The traffic on the interconnection network can be high.

This paper concentrates on the implementation and testing of the Workload Partitioning Segment. Description of the Site Selection and Operation Ordering Segments is provided in [1]. We obtain the necessary samples but are not concerned with the implementation of an optimal sampling scheme. The selection of appropriate sampling technique is beyond the scope of this effort. The interested reader is referred to [3,7,28]. It is further noted that if the database manager superimposes the sampling process with the exeution of the Selection process, often being the first relational operation, the time overhead imposed by the sampling process is nullified.

The following example demonstrates the processes involved during the workload partitioning. Consider a bibliographic relation defined over the following scheme:

R[**Author_Name**, Book_Title, Book_Description,

Call_No, Year_of_Publication].

The keys of the relation are indicated in bold. The relation resides on a multiprocessor system. Consider a user searching for the titles and descriptions of books written by authors whose names are indicated in the Author_List:

$$\text{Author\_List} = \{\text{"Allen"},\text{"Black"},\text{"Clark"},\text{"Dungan"},\text{"Elmer"},\text{"Fields"}\}.$$

To acquire this information the system must perform the following computation:

$$\pi_{\text{Book\_Title,Book\_Description}}\left(\sigma_{\text{Author\_Name} \in \text{Author\_List}}(R)\right).$$

The sequence of relational operations that need to be applied and the intermediate relations created are as follows:

$$\textit{Select}_A(R) \Rightarrow R^1, \quad \text{Project}_B(R^1) \Rightarrow R^2,$$

where A and B are the select condition and the projected attributes, respectively. The Partitioning Segment partitions the workload into disjoint jobs and assigns each job to a processor.

Assume that the number of entries associated with each author name are:

"Allen":	2000 entries,	"Black":	1000 entries,
"Clark":	1000 entries,	"Dungan":	500 entries,
"Elmer":	250 entries,	"Fields":	1000 entries

If the multiprocessor system consists of three nodes and the uniformity assumption is made, the workload might be partitioned into three jobs as follows:

Job 1: "Allen","Black",	3000 entries,	assigned to Node 1
Job 2: "Clark","Dungan",	1500 entries,	assigned to Node 2
Job 3: "Elmer","Fields",	1250 entries,	assigned to Node 3

The above naive workload distribution, however, causes the first node to handle a job more than twice as large as the jobs assigned to each of the other two nodes. We need a balanced allocation (or a close approximation thereof) of workload to processors, even under skewed data inputs. PART achieves this very objective.

PART operates as follows. A random sample of size n is drawn from relation R, to produce relation R_n. PART uses R_n as a representative of R and performs the query operations on the sample R_n, instead of the original relation R. More precisely, PART, performs the operations outlined below:

$$\textit{Sample}(R) \Rightarrow R_n, \quad \textit{Select}_A(R_n) \Rightarrow R_k^1, \quad \textit{Project}_B(R_k^1) \Rightarrow R_k^2$$

PART uses relations R_n and R_k^1 to determine the frequency distribution of the input data set and partitions the workload into three mutually exclusive subsets of approximately equal size. These subsets are then assigned to a particular processor for processing. In our example, PART partitions the workload into the following three jobs:

Job 1: "Allen",	2000 entries
Job 2: "Black","Clark",	2000 entries
Job 3: "Dungan","Elmer","Fields",	1750 entries

The workload of the above three jobs is still not equal. PART divides the load into partitions even finer than the above three. The purpose of this example, though, is to convey the concept behind the PART segment and not to describe in detail the sequence of operations involved. Detailed description is given further in this paper.

The remainder of this paper is organized as follows. The second section contains a literature review of the query optimization efforts and a comparison with our research. Section 3 describes the implementation of the Workload Partitioning Segment. Pseudo-code is included for selected algorithms with primal functionality. The fourth segment provides the mathematical formulation of the developed optimization approach. The behavior of PART is presented in Section 5.

2. Related efforts

There has been extensive research in the area of query optimization, especially for the optimization of the join operation. Join requires more processing power than the other two common relational operations, Select and Project, therefore, Join has attracted particular interest from the scientific community. A survey article by Mishra and Eich, [25], describes the different kinds of joins and the various implementation techniques. In this section, we present the ongoing research in the area of Workload Partitioning for the Join operation. The efforts focus on the interaction of two disciplines, *Selectivity Estimation*, and *Workload Distribution* on the nodes of a multiprocessor system. The Selectivity Estimation provides the required frequency distribution of the participating relations for the heuristic algorithms of the Workload Distribution process.

Selectivity estimation is the ratio of the tuples that satisfy the predicate(s) of the relational operation over the total number of tuples examined. An overview of the statistical profile estimation is given by Mannino, Chu, and Sager [24]. The authors list the most common techniques used to estimate the selectivity of the various relational operations. Muralikrishna and DeWitt [26] evaluate the accuracy of equi-depth histograms to estimate the selectivity of a query. Lynch, [23], presents results on the effectiveness of four selectivity estimators for large bibliographic databases. Lipton, Naughton, and Schneider, [22], estimate the selectivity of relational operations by sampling the target relation until a confidence criterion for the correctness of the estimated selectivity is reached. Hass and Swami, [14] extend the above work, and they develop an "asymptotically efficient" algorithm. The sampling cost of the algorithm becomes equal to the minimum possible sampling cost as the precision requirements become increasingly stringent.

In all the above work, the selectivity estimation stops at the first level of the query execution tree. To obtain the selectivity estimation for the higher levels of the query tree, the researchers use heuristics that operate on the initial estimates. Ioannidis and Christodoulakis, [17], point out that the propagation of the error

associated with the selectivity estimation of the join operation, increases exponentially with the number of the join operations. The error decreases when some accurate information about specific elements on all relations, initial and intermediate ones, is maintained.

In our approach, we also use sampling to obtain the frequency distribution of the underlining relations. Contrary to the relevant efforts that use mathematical models to derive the optimized queries, DOME, executes the relational operations at the current level of the query tree to generate new intermediate relations. These intermediate relations are then sampled to determine an efficient query implementation for the next level of the query tree. In Section 4, we show that by dynamically characterizing the results at each level of the query tree, estimation errors do not grow and do not propagate.

The research efforts in the area of Workload Distribution concentrate on parallelizing various sequential relational operation algorithms, and apply them on parallel machines. Schneider and DeWitt, [29], perform a comparative study of four join algorithms on the Gamma machine [8], namely, Sort-Merge, Simple-hash [9], Grace Hash-Join [20], and Hybrid Hash-Join [9]. Chandeharizadeh and DeWitt, [5] examine the declustering problem, and they develop algorithms for the Gamma machine. The authors presents a methodology to determine the number of optimal relation fragments. The actual fragmentation and the allocation of the fragments to the processors is not examined.

Kitsuregawa and Ogawa, [19], and Hua and Lee, [15,16] present a methodology for the balanced declustering of the workload and distribution of the fragments on the nodes of the SDC machine. Wolf et al., [32], partition the workload associated with the execution of a join operation into two different type regions and allocate the regions to the processors. They propose an algorithm that repartitions the regions, if the allocation of the workload is not even.

In all of these papers, the researchers use heuristics to partition the workload into subsets and then distribute these subsets onto the nodes of the parallel machine. If the workload distribution is not uniform, a bucket tuning phase follows. During the tuning phase the buckets are further subdivided and redistributed among the nodes. The bucket tuning operation continues until a satisfactory level of uniform workload distribution is reached. DOME, on the other hand, acquires the frequency distribution of the participating relations through sampling, and uses the information to determine the bucket ranges in one-shot. The bucket ranges obtained by DOME correspond to a uniform workload distribution, therefore bucket tuning is not required.

Baru and Frieder, [4], and Frieder, [12], present broadcast based and bucket based algorithms for the join workload distribution on the hypercube machine. Omiecinski and Tien, [27], examine the same problem and they propose two hashing algorithms. The authors partition the workload associated with the execution of a relational operation based on statistical information of the participating relations. This information is valid during the first level of the query tree. However, sequence of relational operations modify the statistical information, and the mutation between the real statistics and the estimated ones increase exponen-

tially [17]. The authors do not examine the efficiency of the partitioning algorithms after the first level of the query tree. DOME generates on-the-fly statistical information, at each level of the query tree. The statistics are therefore valid at all levels, and the partitioning of the workload remains efficient.

Our proposed optimizer utilizes techniques developed on other systems, but the novelty of our work relies on the following aspects. DOME uses a statistical algorithm to dynamically partition the input domain of the relational operations. This statistics based approach guarantees a near-uniform processor workload. DOME does not require any initial information of the frequency distribution of the input relations. Lastly, we implemented DOME on the hypercube and we ran tests that show the efficiency of our optimizer.

3. Workload partitioning segment infrastructure

The Workload Partitioning Segment of DOME partitions the workload associated with each query and distributes the tasks to the nodes of the hypercube system in a manner that guarantees near-uniform processor employment. A description of the execution process in presented below:

(1) Receive a query from the database manager. Translate the query into a sequence of relational operations and form a query tree.
(2) Acquire a random sample of all relations involved in the query.
(3) Sort the tuples contained in the sample.
(4) Replicate the sample relations on all nodes of the system. Since the size of the sample is relatively small as compared to the size of the underlining relations, the samples will reside in main memory partitioned over the hypercube nodes.
(5) Examine the frequency distribution of the participating relations to generate buckets that achieve uniform workload distribution. The number of buckets equal the number of nodes of the multiprocessor system.
(6) Use the boundaries of the buckets to partition the entire relations into disjoint segments.
(7) Distribute each segment to each node, and execute the relational operation of the current level of the query tree. Generate intermediate relations. Go to step 5.

In this paper we examine the processes involved during steps 4 to 6. We focus on the manipulations of the sample sets, and we present performance figures for the execution of the relational operations solely on the samples using dynamic verus static partitioning. The Workload Partition segment consists of three modules that handle the above operations.

- *Relational Module*: handles the execution of the relational operations.
- *Partitioning Module*: handles the partitioning of the input workload.
- *Communication Module*: handles the transmission of the various files among the nodes of the system.

3.1 Structures and primitive functions

The characteristics of each relational operation are encapsulated within structure RelationalOperation. The public fields of the RelationalOperation structure are as follows:

- *relR*, *relS*, *relF*: two input relations and one output relation, respectively, that participate in the execution of the current relational operation. For the unary operators (Select and Project) relS is nil.
- *histR*, *histS*, *histF*: histogram files of the two input and one output relation, respectively. For the unary operators (Select and Project) histS is nil. Histogram files contain (key, freq) pairs, while **freq** is the number of tuples in the underlining relation and **key** is the value of the attribute involved with the current relational operations.
- *bucketTable*: table with the bucket boundaries of the input relations. On a system with n nodes, the bucketTable has n entries, one per node. Each entry contains the (lowkey, highKey) pair of the corresponding bucket.
- *selCondition*: conditions of the Select operator.
- *attrR*, *attrS*, *attrF*: the attributes of the tuples from the two input relations that participate in the current relational operations, and the attribute of the output relations. The system maintains the frequency distribution of these three attributes for each relational operation. For the unary operatiors (Select and Project) attrS is nil.

The relation and the histogram files are accessed and updated by the following global functions:

- *tupleFind*(*Relation r*, *Attribute a*, *Value v*): returns the first tuple in relation **r** with value **v** in attribute **a**.
- *tupleNext*(*Relation r*, *Attribute a*): return the next tuple of relation **r**, associated with attribute **a**.
- *tupleAdd*(*Relation r*, *Tuple t*): add tuple **t** in relation **r**.
- *tupleDelete*(*Relation r*): delete the current tuple from relation **r**.
- *histFind*(*Histogram h*, *Value v*): return the (key, freq) entry from the histogram file **h** with key value **v**.
- *histNext*(*Histogram h*): return the next entry from histogram file **h**.

3.2 Partitioning module

The Partitioning module provides a uniform partition of the input workload. It consists of two services, namely:

- *Task Partitioning Service*.
- *Histogram Combination Service*.

The Task Partitioning Service includes routines that partition the workload of the various relational operations into disjoint subsets. The partitioning is performed as evenly as possible, so that all processors require approximately the same time to complete their jobs.

The partitioning operations precede the relational operations. All nodes wait for the partitioning before they can start the execution of the relational operation.

The partitioning routines could have been executed by one node, which then had to broadcast the ranges to the other nodes of the system. In the meantime, all the other nodes would have been idle. This method imposes communication overhead for the broadcasting of the bucket ranges.

A second methodology directs all nodes to perform the partitioning process. The nodes maintain the required information, therefore, there is neither overhead for the transmission of the required histograms, nor for the broadcasting of the bucket ranges. The latter methodology saves the communication time required by the former methodology at the expense of duplicating the work on all nodes. Such duplicate effort is not considered waste in a multiprocessor system. The PART segment adopts the second methodology, because it saves the communication overhead.

During our experimentation we noticed that the partitioning of the input workload into buckets with the same volume of tuples did not provide sufficient speedup as we increased the number of processors. Our analysis showed that the workload of the join operation is not linear proportional to the volume of the tuples involved in the operations. Our experimental results confirm our analytical findings. Detailed description of this Join Workload Skew phenomenon is given in [2]. In this paper we only describe the phenomenon in accordance with a simple example, and give the formula we use for the derivation of the Join workload.

Consider two relations, A, and B, with one attribute each. Assume that the tuples of the relations are as follows:

$$A = \{0, 1, 1, 1, 2, 3, 4, 5\}; \text{ and } B = \{0, 1, 2, 3, 4, 5\}.$$

We join these two relations on a two node system. A naive partitioning of the input workload will generate the following two buckets:

$$\text{bucket } 1: A = \{0, 1, 1, 1\}, B = \{0, 1\}$$

$$\text{bucket } 2: B = \{2, 3, 4, 5\}, B = \{2, 3, 4, 5\}$$

We assign the two buckets to nodes 1 and 2, respectively. Under this partitioning scheme the two nodes will join the following pair of tuples:

$$\text{Node } 1: A \bowtie B = \{(0,0),(1,1),(1,1),(1,1)\}$$

$$\text{Node } 2: A \bowtie B = \{(2,2),(3,3),(4,4),(5,5)\}$$

The volume of data to be processed is equal on both nodes, however, the workload of node 2 is greater than nodes 1. The reason for this skewness is the high overhead of the **find** operation as compared with the **next** operations. Node 1 joins several tuple pairs with the same value, 1, therefore, node 1 uses the global function **tupleNext** several times to retrieve the tuples with value 1. Node 2, on the other hand, joins tuples with different value, therefore, uses the global function **tupleFind** to retrieve these tuples. In any computer system the **find** operation is more time consuming that the **next** operation. To alleviate this Join Workload phenomenon we introduce the *Skew Adjustment* parameter. The skew adjustment parameter is represented by the greek letter β, and is dependent on the configura-

tion of the target hardware system that executes the database operations. For our hardware platform, Intel i860, the Skew adjustment parameter is 3.

It is shown in [2] that the workload of the join operation is related to the frequency distribution of the input relations by the following formula:

$$W = C \times \sum_{i \in R} \sum_{j \in S} freq_i \times (freq_j + \beta)$$

where,

C	constant
R, S	two input relations
i, j	pair of tuples from relations R and S, respectively with the same value on their joining attribute
$freq_i$	frequency of tuples with value i in their joining attribute
β	Skew adjustment parameter

3.2.1 Task partitioning service

The Task Partitioning Service partitions the workload of the relational operations. The main loop of operations is performed by function TaskDetermine. Function TaskDetermine scans the histogram files of the participating input relations for each relational operation and calculates the workload associated with the execution of the particular operation. Subsequently, TaskDetermine, partitions the workload into disjoint buckets, one for each processor, depending on the relational operation under study (Select, Project, Join) and the dimension of the allocated cube. The buckets are allocated to the nodes of the system for processing.

For the Project operation, TaskDetermine scans the histogram file of one of the projected attributes of the input relation. For each entry of the histogram file, TaskDetermine adds the frequency of the tuples, to calculate the number of tuples that undergo processing. The workload of the Project operation is a linear function of the number of tuples that undergo processing.

For the Select operation, TaskDetermine requires one histogram file for the first predicate of each minterm. We describe the format of the SelectCondition in Section 3.3.1. Function TaskDetermine scans each histogram file and calculates, the number of tuples within the bucket of each minterm. Function TaskDetermine then adds all these numbers to obtain the number of tuples that undergo processing. The workload of the Select operation is a linear function of the number of tuples that undergo processing during the execution of the Select operation.

For the Join operation, TaskDetermine traverses the histogram files for the two input relations, R and S. For each subset of R with tuples having the same value on their joining attribute, multiply the frequency of the R tuples by the frequency of the corresponding S tuples adjusted by weight β. These results are then added to calculate the workload of the Join operation.

The workload associated with each relation operation is represented by variable **workload**. This variable is then divided by the size of the allocated cube to determine the optimum workload that each processor should handle, indicated by

variable workloadPerNode. The pseudo-code description of TaskDetermine is presented below.

```
void TaskDetermine(RelationalOperation rop, Operation op, int cubeSize)
{
    workload = 0;
    histR = rop.histR;
    histS = rop.histS;
    bcTbl = rop.bucketTable;
    sc = rop.selCondition;
    switch (op)
        {
        case join:
            rFreq = histFirst(histR);
            while (rFreq)
                {
                    sFreq = histFind(histS, histR.key);
                    workload + = rFreq * (sFreq + β);
                    rFreq = histNext(histR);
                }
            workloadPerNode = workload/cubeSize;
            for (i = 0; i < cubeSize; i + + )
                joinRange(histR, histS, bcTbl[i], b, workloadPerNode);
            break;
        case project:
            rFreq = histFirst(histR);
            workload + = rFreq;
            workloadPerNode = workload/cubeSize;
            for (i = 0; i < cubesize; i + + )
                projectRange(histR, bcTbl[i], workloadPerNode);
            break;
        case select:
        for (i = 0; i < minterms; i + + )
            {
                currKey = sc[i].key;
                rFreq = histFind(histR, currKey);
                while (rFreq)
                    {
                        workload + = rFreq;
                        rfreq = histNext(histR);
                    }
                workloadPerNode = workload/cubeSize;
                for (j = 0; j < cubeSize; i + + )
                    selectRange(histR, sc[i], bucket[j], workloadPerNode);
            }   /* for loop */
        break;
    }   /* switch (op) */
}
```

Function TaskDetermine repeatedly calls the ranging functions, *joinRange*, *projectRange*, and *selectRange*, once per bucket. The ranging routines set the bounds of the workload buckets. There are as many buckets as the nodes on the multiprocessor system. At each invocation, TaskDetermine provides to the ranging

functions the lower limit of the bucket and variable workloadPerNode. Each ranging function sets the upper limit of the bucket and the number of tuples that will be generated had the appropriate relational operation been applied on the tuples indicated by the range of the bucket. Initially, the lower limit of the first bucket is the first tuple of the relation. At each successive invocation, the lower limit of the current bucket is the tuple following the upper limit of the previous bucket.

The ranging functions initialize the workload associated with each to zero. These functions increase the workload by the appropriate number of tuples as they scan the histogram files of the relations that participate in the execution of the corresponding operation. The process stops when the value of the workload becomes equal or exceeds the optimum workload, indicated by variable workload-PerNode. The ranging functions then fill the upper bound of the bucket with the current key and frequency.

The ranging functions ensure that the bucket boundaries do not generate duplicate load on the nodes of the systems and cover completely the input domain. Thus, during the join operation the JoinRange function generates buckets with boundaries that always coincide with a key value boundary of the input relations. Assume, for example, two relative uniform relations R, and S, to join on a two node systems. Let the value of the joining attribute be in the range 1 to 10. The joinRange will ensure that the bucket boundary of the inner relation S will reside on key value 4, 5, or 6, and not within the range of consecutive key values, (4-5), (5-6). The bucket boundary of the outer relation R can reside anywhere withing the range [1–10]. Similar restrictions apply for Project.

```
void projectRange(Histogram hist, bucket bc, int workloadPerNode)
{
    workloadAllocated = 0;
    (key, freq) = getLowBound(bc);
    while (workloadAllocated < workloadPerNode)
        {
            workloadAllocated + = freq;
            (key, freq) = histNext(hist);
        }
    setHighBound(bc, hist.key, freq);
}

void selectRange(Histogram hist, Condition cond, Bucket bc, int WorkloadPerNode)
{
    workloadAllocated = 0;
    (key, freq) = getLowBound(bc);
    while (workloadAllocated < workloadPerNode)
        {
            freq = histFind(hist, key);
            if (key satisfies cond)
                workloadAllocated + = freq;
            (key, freq) = histNext(hist);
        }
    setHighBound(bc, hist.key, freq);
}
```

```
void joinRange(Histogram histR, Histogram histS, Bucket bc, int β, int workloadPerNode)
{
    workloadAllocated = 0;
    tuplesAllocated = 0;
    (rKey, rFreq) = getLowBound(bc);
    while (workloadAllocated < workloadPerNode)
        {
            while (rFreq)
                {
                    rKey = histR.key;
                    sFreq = histFind(histS, rKey);
                    workloadAllocated += sFreq + β;
                    rFreq = rFreq - 1;
                    tuplesAllocated += sFreq;
                }
            (rKey, rFreq) = histNext(histR);
        }
    setHighBound(bc, histR.key, rFreq);
}
```

3.2.2 Histogram combination service

The relational operation module creates histogram files along with the creation of the intermediate relations. Each node generates its own histogram file for the part of the relation it creates. All the individual histogram files need to be combined into one histogram that describes the whole relation. The Histogram Combination service performs this very operation.

The combination process starts by collecting all the partial histogram files from the other nodes, and sends its partial histogram to all the other nodes. Subsequently, the process scans the partial histograms and creates the histogram for the whole relation, by combining all the collected partial information.

The histogram files are small as compared to the relation files; the overhead imposed by the histogram file combination module is negligible as compared to the time requirements for the execution of the relational operations. Our experimental results confirm this statement. The Histogram Combination service uses the function relationTransfer to transfer all the partial histogram files among the nodes of the system. Function relationTransfer is presented in the Communication Module section.

3.3 Relational module

The relational module includes services that support the execution of the relational operations, namely:

- *Relation Accessing Service*
- *Relation Operation Service.*

3.3.1 Relation accessing service

We used a commercial product called C-Index, developed by Trio Systems Inc., California, as our B^+-tree toolkit. C-Index is written in the C language and is a complete implementation of a B^+-Tree Indexed Sequential Access Method (ISAM)

with extended capabilities for variable length data handling. The user can use C-Index to create and maintain multi-keyed records. Specific features of C-Index include:

- Variable length key and data.
- Fully automated multi-key storage.
- Single and multiple-keyed access to data.
- Addition and deletion of keys in any order.

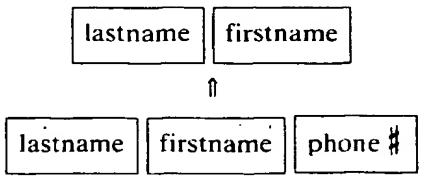
C-Index consists of a set of functions that are linked into the application program. The functions can open, close, or create a relation file, and add, delete, and find entries, in addition to other operations. All function calls to C-Index must supply a pointer to a structure of type *CFILE* which holds information about the relation on which the function is operating.

3.3.2 Relation operation service

PART supports the three basic relational operations, Select, Project and Join. Each relational operation is performed in parallel by all nodes of the hypercube system. The nodes operate on disjoint subsets of the input relations. More specifically, the actions performed for the three relational operations are as follows.

Parallel join is implemented by the *parallelJoin* function. Both relations that undergo join need to be sorted on the values of their joining attributes. The tuples of the outer relation, R, are accessed sequentially and the inner relation, S, is probed to obtain a tuple with the same value in its joining attribute. These two tuples are joined and the composite tuple is added to the resulting relation. Subsequently, relation S is sequentially accessed to obtain all other tuples with the same value in the joining attribute and join them with the same tuple of relation R. The sequential scan of relation S stops when the algorithm finds a tuple with a different value in the joining attribute. Function *parallelJoin* then obtains the next tuple of relation R and the above procedure starts again.

Parallel project is implemented by the *parallelProject* function. The input relation must be sorted on an index which is the concatenation of the attributes that will be projected. For example consider the following projection:



Let the structure generated by the concatenation of the *lastname* and *firstname* attributes be called *lfname*. The *lfname* structure is the sorting index of the relation. The parallelProject function scans the relation in ascending order of the lfname field until the upper bound of the bucket is reached. The B^+-tree indexing scheme used in DOME allows relations to be sorted on more than one index concurrently. For each tuple encountered, function parallelProject projects the

tuples lastname and firstname, and adds the generated tuple to the resulting relation only if a duplicated tuple does not exist already.

Parallel select is implemented by the *parallelSelect* function. The Select condition is in the sum of minterms form. The first predicate of a minterm is the one that determines the distribution of the tasks to the processors. The processors use the attribute associated with the first predicate of a minterm as an index to access the tuples from the input relation that satisfy the current predicate.

The tasks performed during the Select operation depend on the *Select Condition*. The Select Condition is represented in the SUM of minterms form

$$\text{Select Condition} = (A_1 \wedge A_2 \wedge \cdots A_n) \vee (B_1 \wedge B_2 \wedge \cdots B_k) \vee \cdots \vee (Z_1 \wedge Z_2 \wedge \cdots Z_l)$$

$A_1, A_2 \ldots A_n$ are *predicates* that need to be satisfied. A product of predicates forms a *minterm*.

Consider, for example, the bibliographic database defined earlier in this chapter. Assume that a user is searching for entries of books written by authors "Allen", or "Black", and with the year of their publication being prior to 1970. The following computation is required:

$$\sigma_{\text{Author\_Name} \therefore \text{Year\_of\_Publication} \therefore \text{Select\_Condition}}(R).$$

Symbol $\therefore$ indicates that the left term of the expression should satisfy the right term. In our particular example, the values for the Author_Name and the Year_of_Publications should satisfy the particular *Select_Condition*:

$$\text{Select\_Condition} = (A_1 == \text{"Allen"} \vee A_5 \leq 1970) \vee (A_1 == \text{"Clark"} \vee A_5 \leq 1970)$$

Variables A_1 and A_5 represent the first and fifth attribute of relation R respectively.

Function *parallelSelect* examines if every tuple obtained with the above function satisfies all the other predicates of the current minterm. If the tuple does satisfy all the predicates, then the tuple becomes a candidate for the resulting relation. If, however, the tuple just derived satisfies the predicates of any previous minterm, than this tuple has already been entered in the resulting relation, and does not need to be added again. Function parallelSelect will first examines the satisfiability of the previous minterms by the current tuple and only then will add it is the resulting relation.

Consider the following example. Let a bibliographic relation be defined over the schema:

$$R[\textbf{Call\_No}, \text{Author\_Name}]$$

where the key is indicated in bold. Let relation R consist of the following six tuples:

[1,"Allen"], [2,"Black"], [3,"Clark"],
[4,"Dungan"], [5,"Elmer"], [6,"Fields"]

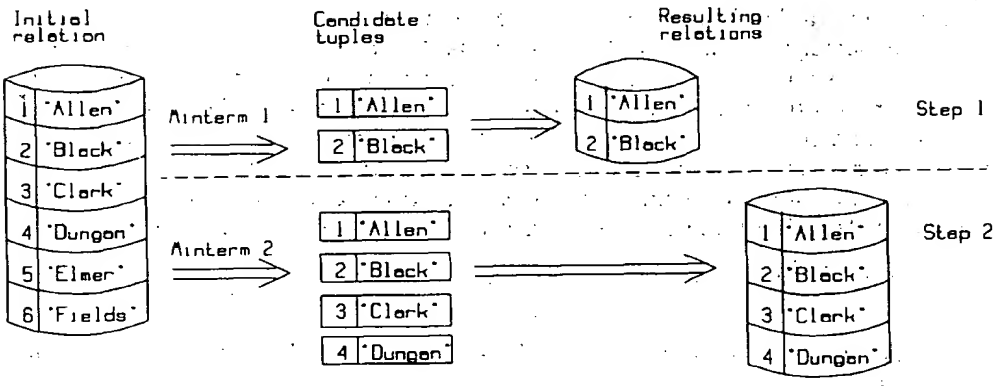


Fig. 1. Sequence of operations for the evaluation of two minterms, during a Select operation.

Let the select condition be:

$$Select\_Condition = (\text{Call\_No} \in [1,2]) \wedge (\text{Call\_No} \in [1,4])$$

During the evaluation of the first minterm, function *parallelSelect* will add to the resulting relation the tuples:

[1,"Allen"], [2,"Black"].

During the evaluation of the second minterm, function *parallelSelect* will identify, as candidate, the resulting tuples:

[1,"Allen"], [2,"Black"], [3,"Clark"], [4,"Dungan"].

The first two tuples, however, already exist in the resulting relation, therefore, the algorithm does not add them again. Function *parallelSelect* will enter only the third and fourth tuple. The above sequence of operations is presented in Fig. 1.

All the above functions generate histogram files on some predetermined attributes of the resulting relation. We keep histogram information on attributes that participate in the follow-up relational operation of the query tree.

The pseudo-code of the relation operation module functions is presented below:

```
int parallelProject(RelationalOperation rop, int node)
{
    tuples = 0;
    partition = rop.bucketTable[node];
    lowBound = partition.lowBound;
    highBound = partition.highBound;
    index = lowBound;
    tupleR = tupleFind(rop.relR, rop.attrR, index.key);
    while (index < highBound)
        {
            tupleF = project(rop.tupleR);
            if (tupleF ∉ rop.relF)
                tupleAdd(rop.relF, tupleF);
            updateHistogram(rop.histF, tupleR.key);
            index = tupleNext(rop.relR, rop.attrR);
```

```
                tuples++;
            }
        return tuples;
}

int parallelSelect(RelationalOperation rop, int node)
{
    tuples = 0;
    minterm = rop.minterm;
    for (i = 0; i < elements(minterm); i++)
        {
            partition = rop.bucketTable[node];
            attrR = rop.attrR;
            lowBound = partition.lowBound;
            highBound = partition.highBound;
            index = lowBound;
            tupleR = tupleFind(rop.relR, attrR, index.key);
            while (index <= highBound)
                {
                    if (tupleR satisfies all predicates of minterm[i] &&
                        !(tupleR satisfy any predicates of minterm[0,.i-1]))
                        {
                            tupleAdd(rop.relF, tupleR);
                            tuples++;
                            updateHistogram(rop.histF, tupleR.key);
                        }
                }
        }
    return tuples;
}

int parallelJoin(RelationalOperation rop, int node)
{
    tuples = 0;
    partition = rop.bucket[node];
    lowBound = partition.lowBound;
    highBound = partition.highBound;
    index = lowBound;
    tupleR = tupleFind(rop.relR, rop.attrR, index.key);
    tupleS = tupleFind(rop.relS, rop.attrS, index.key);
    while (index < highBound)
        {
            tupleS = tupleFind(rop.relS, rop.attrS, index.key);
            while (tupleR.attribute[rop.attrR] == tupleS.attribute[rop.attrS])
                {
                    tupleF = join(tupleR, tupleS);
                    tupleAdd(rop.relF, tupleF);
                    updateHistogram(rop.histF, tupleR.key);
                    tuples++;
                    tupleNext(rop.relS, rop.attrS);
                }
            index = tupleNext(rop.relR, rop.attrR);
        }
    return tuples;
}
```

3.4 File communication module

The File Communication module handles the transmission of the relations among the nodes of the hypercube system. The File Communication module achieves full mirroring of the participating relations on the local file systems. The File Communication module is implemented by function **relationTransfer** which transmits and receives files among the nodes. The operation of function relationTransfer is explained below and in accordance with Fig. 2.

Let a four node cube be the host multiprocessor system. The nodes have executed a relational operation and they have created four partial relations. Nodes 0,1,2, and 3 have created the first, second, third, and fourth part of the resulting relation, respectively. All these partial relations need to be transmitted among the nodes, so that every node maintains all the parts of the resulting relation. During the first step of the algorithm the pairs of nodes 0,1 and 2,3 communicate concurrently, and they exchange their partial relations. After the first step, nodes 0, and 1 maintain the first and second parts of the resulting relation, while nodes 2, and 3 maintain the third and the fourth parts. During the second step of the algorithm, the pairs of nodes 0,2 and 1,3 communicate concurrently, and they exchange their augmented partial relations. After the dth step, where d is the dimension of the cube; all nodes contain all parts of the resulting relation.

```
void relationTransfer(int node, Relation outRel, Relation inRel, int cubeSize)
{
    cubeDim = log2(cubeSize);
    inRel[node] = outRel;
    for (i = 1; i < = cubedim; i - + )
        {
            curBit = bit(node, i);
            outNode = grayCode(node, cubeDim - i);
            cubeNum = node / cubeSize;
            rels = cubeSize/2;
            cubeStart = cubeNum * cubeSize;
            cubeMid = cubeNum * cubeSize + rels;
            for (j = 0; j < rels; j + + )
                {
                    if (!curBit)
                        {
                            sendRel(inRel[cubeStart + j], outNode);
                            recvRel(inRel[cubeMid + j]);
                        }
                    else
                        {
                            sendRel(inRel[cubeMid + j], outNode);
                            recvRel(inRel[cubeStart + j]);
                        }
                }
        }
}
```

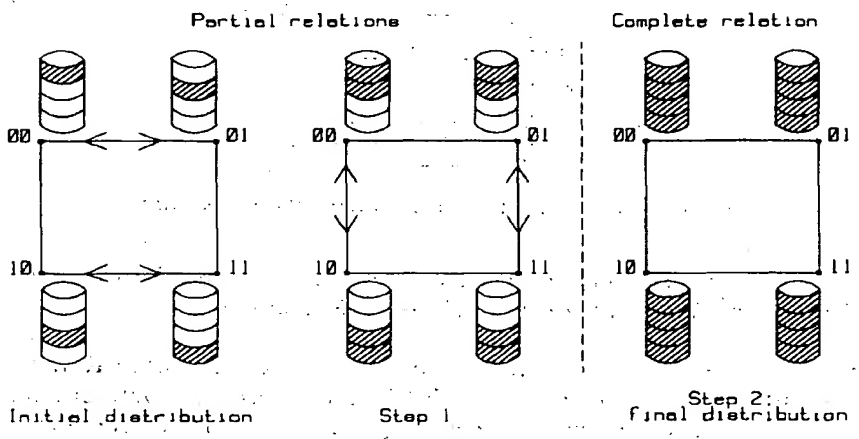


Fig. 2. Steps of the relationTransfer function.

4. Mathematical foundation of the sampling methodology

In this chapter we provide the mathematical justification for the sampling optimization environment used by DOME. Ioannidis and Christodoulakis, in [17], showed that initial statistical information is valid during the first level of the query tree. For each additional operations the existing optimizers manipulate formulas that describe the parameters of the participating relations to develop new formulas with the frequency distribution of the new relations. The frequency distribution described by the optimization formulas differ from the ones derived had the appropriate transformations being applied on the original relations. The difference increases exponentially as the number of relational operations increases.

DOME, on the other hand, applies the sequence of relational operations of a query on the samples of the participating relations. DOME does not manipulate formulas. We prove that relational operations applied to samples of the original relations generate results that can still represent precisely their corresponding unsampled relations. This statement connotes that the application of the same query operators on sampled and unsampled relations generate relations with similar frequency distribution. This property is essential since the optimizer uses the samples, original and intermediate, as an accurate image of the target relations, to determine an efficient bucket ranging. Specifically, we will prove the following *Relational Sampling* theorem.

Theorem 4.1: Relational Sampling Theorem. *Let R_n and S_m be the relations obtained after sampling n and m tuples from two relations R and S. The sample sizes n, and m, can be the same. The relation created when one of the Select or Project operations is applied on R_n (or S_m), or the Join operation applied on R_n and S_m,*

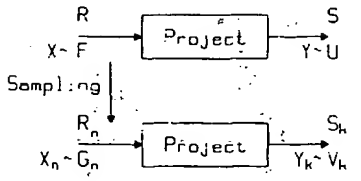


Fig. 3. DOME Optimization processes during Project operation.

converges to the relation that would had been generated had the same relational operation been applied on the original relation(s) R (and / or S).

Definition. One relation R converges to another relation S when R and S have the same number of attributes and the distribution of each attribute of relation R converges to the distribution of the corresponding attribute of relation S. The distribution of an attribute is represented by a random variable.

The consequence of the above definition is that convergence between relations is equivalent to convergence between the random variables that represent the attributes of the relation. Consider for example the Project operation and the sequence of processes indicated in Fig. 3. The processes identified in this figure are followed by DOME during the optimization procedure.

An input relation R undergoes Projection. The Project operation generates relation S. Let random variable X describe the frequency distribution of an attribute under interest, and let X follow distribution function F. The Project operation creates a new random variable Y with distribution function U.

DOME performs sampling of R and generates a sample relation R_n. Relation R_n undergoes Projection, and the resulting relation is S_k. The frequency distribution of X_n is G_n, and the Project operation generates random variable Y_k with frequency distribution V_k.

The Relational Sampling theorem states that relations S and S_k have the same frequency distribution, or equivalently, distributions U and V_k *converge with probability one*. We later define mathematically the premise "converge with probability one."

To prove the Relational Sampling theorem we use the Glivenko-Cantelli [11] theorem. This theorem states that F and G_n converge with probability one. We extend the converge past the Project operation, and show that U and V_k converge also. Before we present the proof of the Relational Sampling theorem, we provide a short description of the statistical terms used throughout this chapter.

Random variable X has a probability distribution function given by function $F_x(x)$. Take a random sample of R, size n, and create R_n. Through this sampling, we obtain a random sample of X represented by its order statistics $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$. The order statistics of X follow distribution function $G_n(x)$. This distribution function, $G_n(x)$, is defined for all real numbers x. $G_n(x)$ is the proportion of sample values which do not exceed the number x. Thus, $G_n(x)$ is a step function which increases by the amount $1/n$ at its jump points which are the

order statistics of the sample. The empirical distribution function of $G_n(x)$ is defined symbolically in (4.1).

$$G_n(x) = \begin{cases} 0, & \text{if } x < X_{(1)} \\ \dfrac{k}{n}, & \text{if } X_{(k)} \le x < X_{(k+1)}, \quad \text{for } k = 1,2,\ldots,n-1 \\ 1, & \text{if } x \ge X_{(n)} \end{cases} \tag{4.1}$$

The Glivenko-Cantelli theorem identifies the relation between $G_n(x)$ and $F_x(x)$.

Theorem 4.2: Glivenko-Cantelli theorem [11]. *$G_n(x)$ converges uniformly to $F_x(x)$ with probability one; that is:*

$$P\left[\lim_{n\to\infty} \sup_{-\infty<x<\infty} |G_n(x) - F_X(x)| = 0\right] = 1 \tag{4.2}$$

The Glivenko-Cantelli theorem holds for any two functions, $F_x(x)$ and $G_n(x)$, the former continuous and the later discrete, related by formula (4.1). This very statement is the only restriction that must hold for two random variable to converge. Therefore, during the following analysis, whenever we need to show convergence among two relations, we will prove the above statement for their corresponding random variables.

Let us now proceed with the first part of Theorem 4.1 that examines the effect of the unary relational operations. A relational operation, $p(\cdot)$ (Select, Project), is applied to the sample relation R_n. A new relation is created, called Q_k. Equivalently, a function $q(\cdot)$ is applied to the random variable X. The distribution function of the target attribute changes; let it now be represented by function $V_k(y)$. Let also $U_y(y)$ be the distribution function of the target attribute when the same relational operation is applied on the original relation R. The above sequence of operations is modeled as:

$$F_X(x) \xrightarrow[q(X)]{p(R)} U_Y(y)$$

sampling ↓

$$G_n(x) \xrightarrow[p(R_n)]{q(X)} V_k(y)$$

We must prove that $V_k(y)$ converges to $U_y(y)$. Since the Glivenko-Cantelli theorem states that converge exists between any two relations, the former continuous and the later discrete, related by formula (4.1), we must show that $V_k(y)$ is related to $U_y(y)$ by a formula similar to (4.1).

We stated before that the application of a relational operation on a relation modifies the probability distribution function of its attributes. Furthermore, this modification is the effect of the application of a function $q(\cdot)$ on the random variable that represents the particular attribute. The application of function $q(\cdot)$, transforms variable X into variable Y. This new random variable $Y = q(X)$ has a new distribution function $U_x(x)$.

The function of a random variable is a mapping from one set of real numbers, $\Re$, to another, $\mathscr{P}$. Every value x_i of the sample $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$ from set $\Re$, maps through function $q(\cdot)$ on a value y_i of the set $\mathscr{P}$. This mapping is pictorially shown in the adjacent Fig. 4.

Let $\Re$ and $\mathscr{P}$ be the domain and range of function $q(\cdot)$, respectively. Let $\mathscr{L}$ be the set of values obtained from the random set. Set $\mathscr{L}$ is a subset of $\Re$. Since every value of set $\Re$ maps onto set $\mathscr{P}$ through function $q(\cdot)$, every value of subset $\mathscr{L}$ maps onto a corresponding set $\mathfrak{C}$, which is a subset of $\mathscr{P}$. We arrange the values of set $\mathfrak{C}$ in increasing order and obtain the order statistics of Y, $Y_{(1)}, Y_{(2)}, \ldots, Y_{(k)}$. The distribution function of $Y_{(1)}, Y_{(2)}, \ldots, Y_{(k)}$ is now given by formula (4.4).

$$V_k(y) = \begin{cases} 0, & \text{if } y < Y_{(1)} \\ \dfrac{l}{k}, & \text{if } Y_{(l)} \le y < Y_{(l+1)}, \quad \text{for } l = 1,2,\ldots,k-1 \\ 1, & \text{if } y \ge Y_{(k)} \end{cases} \tag{4.4}$$

Formula (4.4) relates $V_k(y)$ and $U_y(y)$ and is similar to (4.1). The first part of Theorem 4.1 follows from (4.4) and the definition of the Glivenko-Cantelli theorem.

The join operation is a dyadic operation, where two relations, R and S, are composed to produce the resulting relation, Q. Let X and Y be random variables that represent the joining attributes of relations R and S, respectively. Let Z be a random variable that represents the attribute of relation Q which corresponds to the combination of the two input join attributes. Variables X, Y, and Z are related by function $Z = q(X,Y)$.

Let $F_x(x)$, $H_y(y)$, and $U_z(z)$ be the distribution function of X, Y and Z, respectively. Let R_n, S_m be the relations obtained after sampling n tuples from R and m tuples from S. Let the sample distributions of X and Y be represented by $G_n(x)$, and $J_m(y)$, respectively. By joining relations R_n and S_m, we obtain relation Q_k. Let the distribution of the random variable that represents the target attribute of Q_k be $V_k(z)$. The relation between the above parameters and the sequence of operations applied to obtain them is modeled as:

$$\begin{array}{ccc} F_X(x) \bowtie H_Y(y) & \xrightarrow[q(X,Y)]{p(R,S)} & U_Z(z) \\ \text{sampling} \downarrow & & \\ G_n(x) \bowtie J_m(y) & \xrightarrow[p(R_n,S_m)]{q(X,Y)} & V_k(z). \end{array} \tag{4.5}$$

The goal is again to prove that $V_k(z)$ converges to $U_z(z)$ with probability one. Let $\Re$, $\mathfrak{T}$, and $\mathscr{P}$ be the domain sets and the range set of function $q(\cdot)$, respectively. Let $\mathscr{L}$ and $\mathfrak{C}$ be the set of values of the random samples $G_n(x)$, and $J_m(y)$, respectively. Sets $\mathscr{L}$ and $\mathfrak{C}$, are subsets of $\Re$ and $\mathfrak{T}$, respectively. Every pair of values from sets $\Re$ and $\mathfrak{T}$ maps through function $q(\cdot)$ on set $\mathscr{P}$. Consequently,

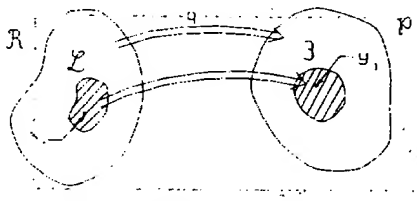


Fig. 4. Set mapping through the application of a function on a random variable.

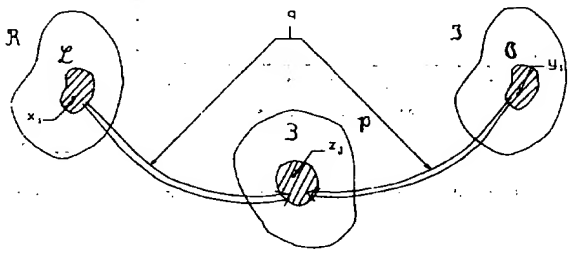


Fig. 5. Set mapping through application of a function on two random variables.

every pair of values form sets $\mathscr{L}$ and $\mathfrak{G}$ maps on a corresponding set $\mathfrak{Z}$, which is a subset of $\mathscr{P}$. This mapping is shown in Fig. 5.

The values of $\mathfrak{Z}$, in increasing order, are the order statistics of Z, $Z_{(1)}, Z_{(2)}, \ldots, Z_{(k)}$. The distribution function of $V_k(z)$ is then given by (4.6).

$$V_k(z) = \begin{cases} 0, & \text{if } z < Z_{(1)} \\ \dfrac{l}{k}, & \text{if } Z_{(l)} \le z < Z_{(l+1)}, \quad \text{for } l = 1,2,\ldots,k-1 \\ 1, & \text{if } z \ge Z_{(k)} \end{cases} \tag{4.6}$$

The second part of theorem 4.1 follows from (4.6) and the definition of the Glivenko-Cantelli theorem. Theorem 4.1 is thus proven. □

5. Experimental evaluation

We present the performance of the PART environment for various input workload and cube sizes. All the following results represent the time required to perform specific relational operations on actual relations on the Intel i860 hypercube system. We ran our experiments on the Intel i860 32 node hypercube system, with 8Mbytes of memory per node. We used a ramdisk environment to emulate local file systems on the nodes.

The ramdisk environment was developed without incorporating any elaborate optimization techniques to decrease the file accessing time. PART's performance

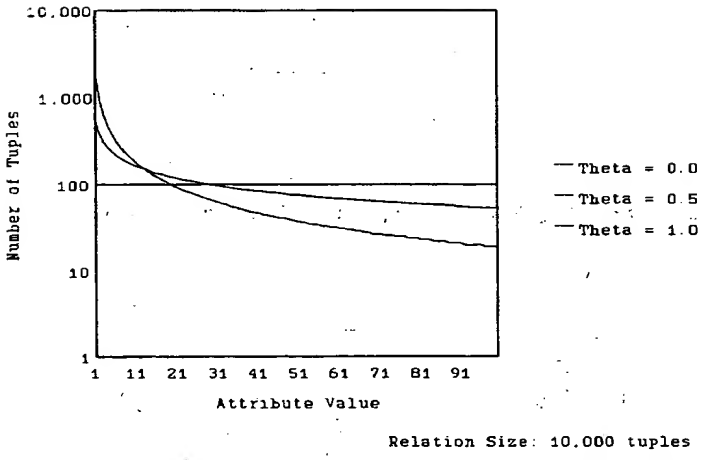


Fig. 6. Frequency distribution of an attribute's value, using Zipf formula for High Skew, Moderate Skew, and Low Skew.

would improve had these optimizations been included, but the improvement would occur flatly across all the various input datasets. It is the purpose of this research to demonstrate the scalability of the PART environment and its improvement over a static optimizer and not its absolute timings.

The input datasets vary by their cardinality and degree of skewness. To examine the speedup achieved by the algorithms developed, we created skewed relations on a particular attribute, based on a Zipf-like distribution [21]. Many other authors, including Lynch and Christodoulakis [23,6], also use the Zipf distribution to model distributions of values that are highly nonuniform. The Zipf distribution is defined as follows: We assume that the domain of the target attribute had D distinct values. The probability p_i that the attribute value of a particular tuple takes on the i^{th} value in the domain $1 \leq i \leq D$ is $p_i = c/i^{1-\theta}$, where $c = 1/\sum_{i=1}^{D}(1/i^{1-\theta})$ is a normalization constant. We assume that each attribute's value is independently chosen from this distribution. Setting the parameter $\theta = 0$ corresponds to the pure Zipf distribution, which is highly skewed, while $\theta = 1$ corresponds to the uniform distribution. Fig. 6 represents the distribution of key values in a relation with 1000 tuples for highly skewed, moderate skewed, and uniform cases.

We performed two sets of tests. During the first set, the evaluation focuses on the sequence of Project-Join operations and includes a comparison of the timings obtained when the dynamic range partitioning is employed. Timings for the Select operation are not provided here because it is parallelized using strictly data partitioning. During the second set, the evaluation focuses on the effect of the Skew Adjustment parameter on the Join operation. We show, by experimentation, that the standard deviation of the partial join execution time improves by an order of magnitude, when the Skew Adjustment parameter is included. Partial job

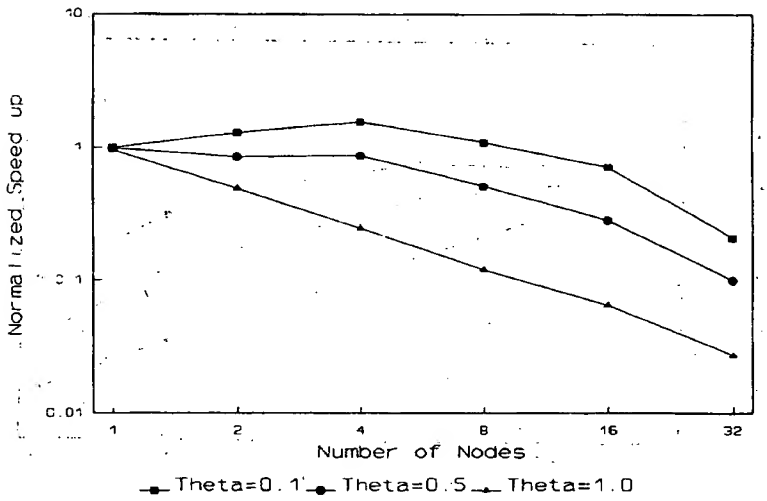


Fig. 7. Normalized Speed-up for Project-Join sequence; the input relations contain 2K tuples each.

execution time with small standard deviation indicates that the disparity between the node's execution time is low; therefore, the utilization of the system is high.

5.1 Project join timings

During the first set of experimentations, we consider two relations defined over the following relations schemes:

R_1[**Lastname**, Address, Phone],

R_2[**Lastname**, Firstname, SSN]

The key of each relation is indicated in bold. We register the time requirements to perform the query

$$\pi_{\text{Lastname,Address}}(R_1) \bowtie \pi_{\text{Lastname,Firstname}}(R_2).$$

The initial sample relations are of two different sizes, 1K, and 2K. Relations were generated using a Zipf-type distribution formula for the value of the key attribute. The values of the other attributes were randomly generated using a uniform value attribute. The tuples of each relation were horizontally partitioned across the nodes in a uniform manner.

In Figs. 7 and 8, we show the normalized speedup for the process of the 'Project-Join' operation sequence. *Speedup* is defined as the ratio of the query execution time with *Static* over *Dynamic* ranging. During dynamic ranging, DOME produces a histogram file along with the generation of all intermediate relations that express the frequency distribution of the corresponding relation. DOME uses the information incurred in the histogram file to partition the workload into equal,

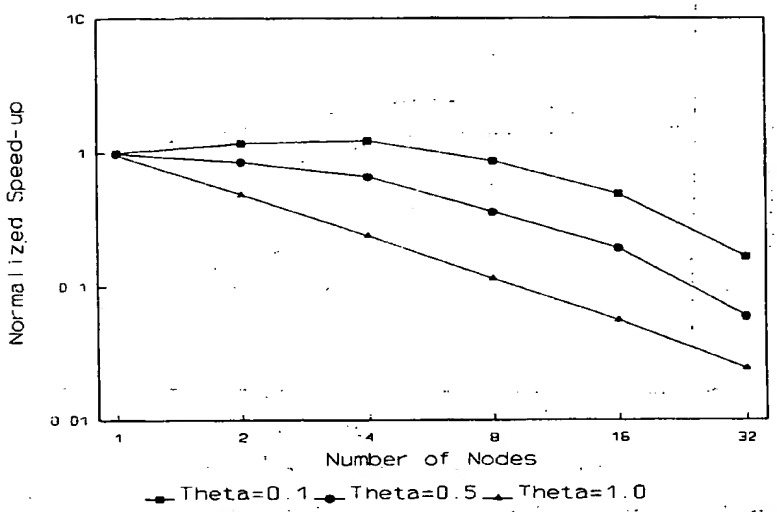


Fig. 8. Normalized Speed-up for Project-Join Sequence: the input relations contain 1K tuples each.

in terms of processing time, buckets. During static ranging, DOME does not keep any statistical information on the distribution of the participating relations and partitions the workload assuming a uniform distribution. *Normalized Speedup* (*NS*) is the ratio of the speedup over the number of system nodes. In ideal cases, the normalized speedup should be one, or close to one, as the number of nodes increase. Fig. 7 presents the NS with input relation size of 2K tuples, and Fig. 8 with input relation size of 1K tuples.

We observe that for small number of nodes, 4 nodes and under, the NS becomes greater the one. This behavior is explained as follows. The workload per node is inversely proportional to the number of nodes that execute the relations operation. When the number of nodes decreases, the workload per node increases. Consequently, for small systems (size 2,4 nodes) the effort to add a new tuple into the resulting relation is higher than large systems. This statement implies that, all other parameters considered equal, small systems experience higher impedance to execute a query than large systems.

The NS decreases above 4 nodes, and gets less than one with systems of 8 nodes and higher. This value of the NS factor for high systems, is attributed to the small cardinality of the input relations. The communication and synchronization overhead associated with the processing of such small datasets by large systems (8 nodes and up), hinder the performance savings that the multiprocessor system provides. We observe from the figures, that as the size of the input relations increase from 1K to 2K, the NS increase also. We can therefore claim, that with bigger input relations, in the range of gigabytes of data, the NS would exhibit less decrease.

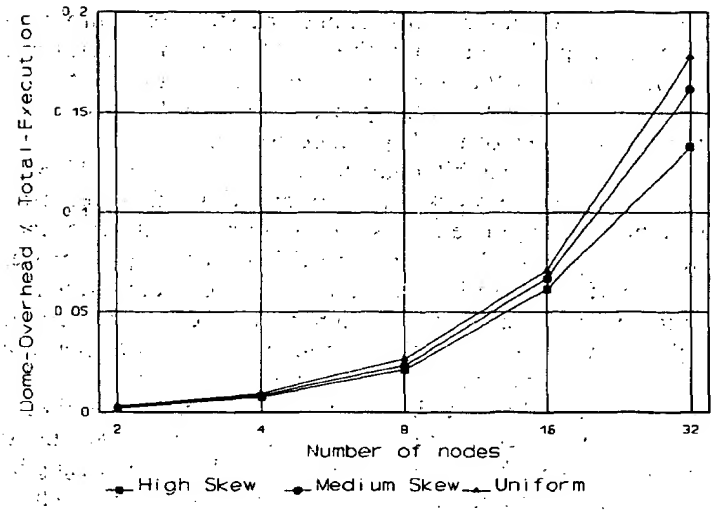


Fig. 9. DOME-Overhead/Total-Execution; 2K input relations.

Figs. 9 and 10 show the ratio between the overhead of DOME versus the total execution time. We examine the Project-Join sequence with Dynamic Ranging. The operations involved during the Project-Join sequence are: Histogram Generation and Bucket Derivation (*HGBD*), *Project*, *Join*, and File Transfer (*FT*). We consider the HGBD as the overhead of the DOME environment. This overhead

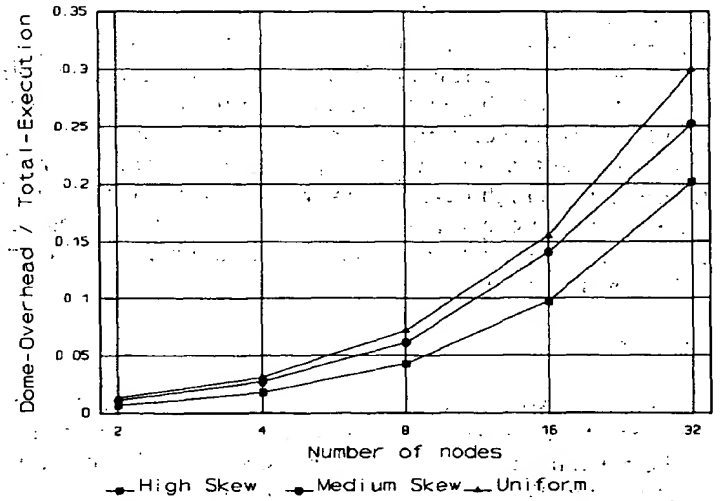


Fig. 10. Dome-Overhead/Total-Execution; 1K input relations.

includes the processes to produce and examine the statistical information of the generated relations and further partition the workload into uniform buckets. Join and Project are the processes that correspond to the execution of the associated relational operations. File transfer is the process that handles the transmission of the relations among the nodes of the hypercube system.

When the input relations exhibit high degree of skew, the HGBD operation enables DOME to divide the workload into partitions that require the same processing time. When the input relations are uniform, any generic static partition algorithm divide the input workload into partitions with equal processing time requirements. The HGBD operation is, therefore, pure overhead when the input relations are uniform, and degrades, although negligible, the performance of the system. With highly skewed input relations, though, the HGBD operation provides uniform workload partitioning and supports system scalability.

The graphs demonstrate that the time requirements for the overhead operation (HGBD) is small as compared to the total execution time. As the size of the hypercube system decreases, the overhead operations becomes negligible, as compared to the time requirements of the total execution time. This seemingly alarming situation, since this indicates the lack of system scalability, is caused by the fact that the samples must now be collected from a large number of nodes. However, the relations are kept at a constant size. Clearly in a production system, the size of the system used would be matched to the volume of data processed. Hence, this growth in the ratio of the overhead would not occur. This is validated by that, that as the size of the input relations increases (from 1K to 2K), the overhead/total-execution ratio decreases. That is, a proper ratio of the size of the system used to the volume of the data processed introduces low overhead. Furthermore, for large input relations, the overhead of DOME is small as compared to the execution time of the relational operation sequence.

The factor improvement of the dynamic versus the static ranging scheme is shown in Fig. 11. Factor improvement is the ratio of the dynamic versus the static ranging scheme. When dynamic ranging is employed to execute the Project-Join query on the 2K relations with highly skewed input relations, the response of a 32 node cube system is almost eleven time faster than if the static ranging method was used. The graph indicates that the factor improvement is particularly high when the input relations exhibit a high degree of skew and many nodes of the hypercube system are employed for the execution of the query. The graph does not show the factor improvement associated with uniform input relations. This factor is roughly 0.95 due to the low overhead incurred.

5.2 RS skew adjustment

For the experiments involving the RS-Skew Adjustment parameter (SA), we consider a join operation between two relations. We first partition the Join workload using the SA parameter, β, into buckets, and distribute these buckets to the nodes for processing. We register the join execution time of each individual node. Subsequently, we repartition the join workload without considering the SA

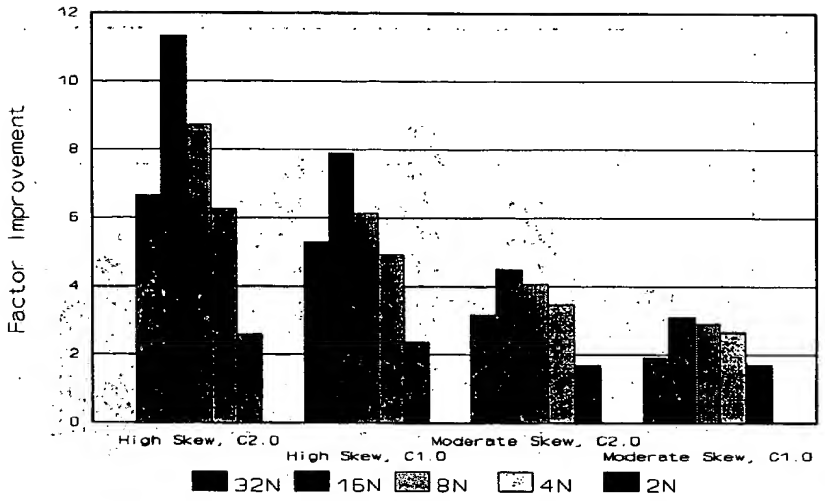


Fig. 11. Factor improvement of the Dynamic versus the Static ranging method.

parameter and distribute these new partitions to the nodes for processing. We register again the join execution times of each node. We finally calculate the standard deviation of the execution time of the nodes with SA and without SA.

The input datasets vary by their cardinality and degree of skew. Relations were generated using the Zipf-type distribution formula for the value of the join key attribute. The values of the other attributes were randomly generated using a uniform distribution. One of the attributes is the key field on which we perform the join. The attributes of both relations are strings, hence, the relatively long computational times. We performed tests for input relation cardinalities 1K and 2K records. For each cardinality case, we join two relations both with high skew, moderate skew, and uniform distribution on their join attribute. Results were obtained using cube sizes of 2, 4, 8, 16 and 32 nodes.

We use the standard deviation of the Join execution times as a metric for the effectiveness of the SA parameter. As indicated in Section 5.1.3, the value of the SA parameter depends on the hardware platform, and, for the i860 hypercube, is approximately equal to the dimension of the currently allocated cube. Let σ_1 be the standard deviation of the Join execution times without the SA parameter and σ_2 the standard deviation with the SA parameter. Fig. 12 and 13 present the ratio σ_1/σ_2 for input relation cardinalities 2K and 1K tuples, respectively. We define this ratio as the *Standard Deviation Factor Improvement* (*SDFI*) of the SA parameter. The figures present the SDFI under various conditions of data skew and hypercube system sizes.

When the input relations are uniform there is no need for SA. Any naive partitioning algorithm efficiently divides the workload of the Join operation and produces jobs with equal time requirements. We observe that in the uniform case

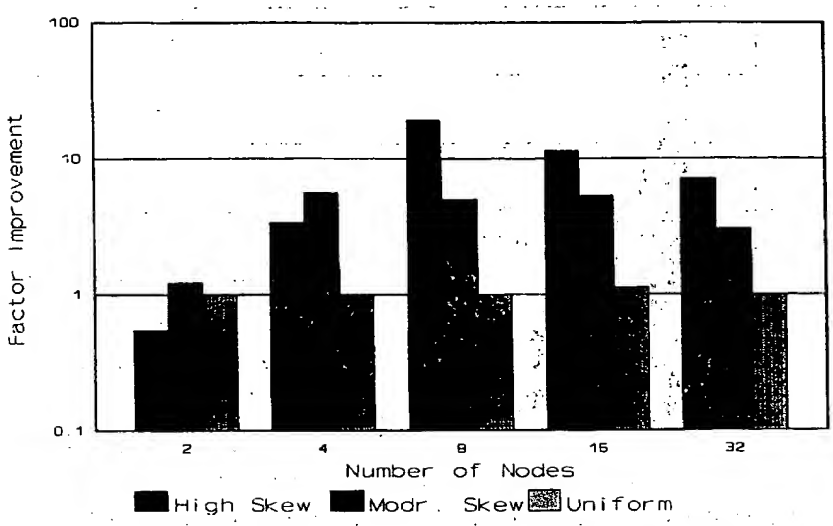


Fig. 12. Standard Deviation Factor Improvement of the SA parameter; Input relations cardinality 2K.

the factor improvement is approximately one. Factor improvement of one indicates that the standard deviation with SA is equal to the standard deviation without SA. In the uniform case, therefore, the SA parameter does not hinder the efficiency of the workload partitioning algorithms.

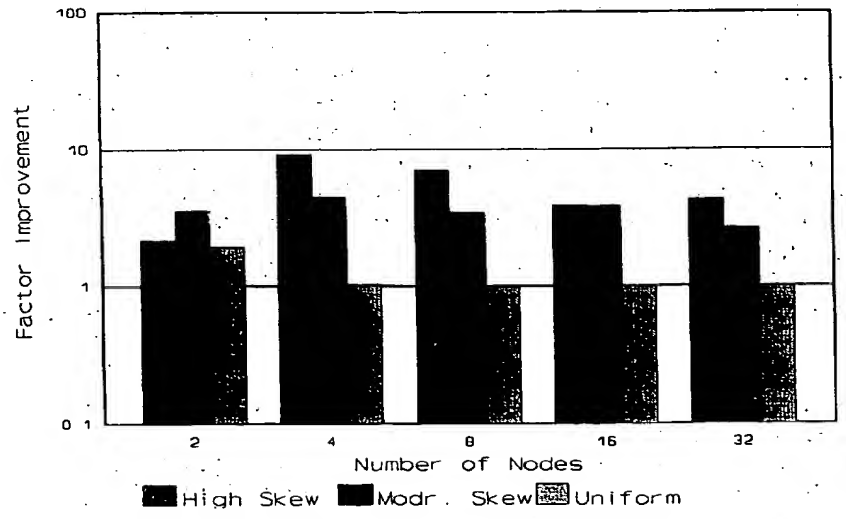


Fig. 13. Standard Deviation Factor Improvement of the SA parameter; Input relations cardinality 1K.

The SDFI, in general, depends on the data skew, and is more profound for highly skew input relations. The factor improvement becomes less than one when the input relations consist of 2K highly skewed tuples and the system size is 2 nodes. During all other situations the factor improvement is higher than one.

The factor improvement increases as the hypercube system size increase. The SA parameter provides a SDFI within an order of magnitude over partitioning without SA for particular cases of relation skewness and system size. SDFI increases as the data skew increases. SDFI reaches a saturation point above which its value declines as the number of nodes are further increased. This saturation point depends on the size of the input relations. For 2K relation cardinalities the saturation point is 8 nodes, while for 1K relation cardinalities the saturation point is 4 nodes. The saturation condition indicates that the system size is high as compared to the size of the input data; therefore, the efficiency of the workload partitioning algorithms declines. Non-efficient partitioning indicates that the processors of the system execute jobs with different time requirements.

6. Conclusion

We presented The Workload Partitioning Segment (PART) of DOME, a query optimizer developed on an Intel i860 hypercube system. PART uses sampling to determine the frequency distribution of the input relations and partition the workload associated with the execution of relational operations in a uniform manner. The partitioning is uniform even when the input relations exhibit a high degree of skew. PART's scalability is proportional to the number of processors used for the execution of the relational operations. The system displays a tenfold performance improvement, when dynamic ranging algorithms of PART are used on highly skewed input relations, over a simple static ranging approach. For future work we will modify PART to operate on heterogeneous environments. The modifications involve tuning of the partitioning routines to accommodate the different processing power of the various nodes. We should also modify the routing module to efficiently transfer files through the new interconnection network.

References

[1] F. Barlos, DOME: Dynamic Optimization on Multiprocessor Engines, A Statistical Approach, Ph.D. Dissertation, Department of Computer Science, George Mason University, March 1993.

[2] F. Barlos and O. Frieder, Join workload partitioning under uniform and skewed input relations, *Parallel Processing Letters* (Aug. 94) 95–104.

[3] V. Barnett, *Sample Survey; Principles and Methods* (Oxford University Press, 1991).

[4] C.K. Baru and O. Frieder, Database operations on a cube-connected multicomputer system, *IEEE Trans. Comput.* 38 (6) (June 1989) 920–927.

[5] S. Chandeharizadeh and D. DeWitt, Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines, *16 VLDB* (1990) 481–492.

[6] S. Christodoulakis, Estimating record selectivities, *Information Syst.*, 8 (2) 105–115.

[7] G.W. Cohran. *Sampling Techniques*. (John Wiley 1963).
[8] D.J. DeWitt et al., The GAMMA database machine project, *IEEE Trans. Knowledge Data Eng*. 2 (March 1990) 44–62.
[9] D.J. DeWitt et al., Implementation techniques for main memory database systems. *ACM-SIGMOD*. (1984).
[10] R. Elmasri and S. Navathe. *Fundamentals of Database Systems* (Benjamin/Cummings, 1989).
[11] M. Fisz, *Probability Theory and Mathematical Statistics*. (John Wiley 1963).
[12] O. Frieder, A parallel database-drive protocol verification system prototype; *Software Practice Experience* 22(3) (March 1992) 245–264.
[13] R. Frost, *Introduction to Knowledge Base Systems* (McMillan, New York, 1986).
[14] P. Hass and A. Swami, Sequential sampling for query size estimation, *ACM-SIGMOD* (1992). 341–350.
[15] K. Hua and C. Lee, Handling data skew in multiprocessor database computers using partition tuning, *17th VLDB* (1991) 525–535.
[16] K. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems, *16th VLDB* (1990) 493–506.
[17] Y. Ioannidis and S. Christodoulakis. On the propagation of errors on the size of join results, *ACM-SIGMOD*. (1991). 268–277.
[18] J.V. Joseph et al., Object-oriented databases: Design and implementation, *Proc. IEEE* 79 (1) (Jan. 1991) 42–64.
[19] M. Kitsuregawa and Y. Ogawa, Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). *16th VLDB* (1990) 210–221.
[20] M. Kitsuregawa, H. Tanaka and T. Moto-Oka, Application of hash to database machine and its architecture, *New Generation Computing* 1 (1983) 63–74.
[21] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. (Reading, MA Addison-Wesley, 1973).
[22] R. Lipton, J. Naughton and D.A. Schneider. Practical selectivity estimation through adaptive sampling. *ACM-SIGMOD*. (1990) 1–11.
[23] C.A. Lynch, Selectivity estimation and query optimization in large databases with highly skewed distribution of column values, *14th VLDB* (1988) 240–251.
[24] M. Mannino, P. Chu and T. Sager, Statistical profile estimation in database systems, *ACM Computing Surveys* 20 (3) (Sep. 1988) 191–221.
[25] P. Mishra and M.H. Eich, Join processing in relational databases, *ACM Computing Surveys*, 24 (1) (March 1992) 63–113.
[26] M. Muralikrishna and D. DeWitt, Equi-depth histograms for estimating selectivity factors for multi-dimensional queries, *ACM-SIGMOD* (1988) 28–36.
[27] E. Omiecinski and Lin E. Tien The adaptive-hash join algorithm for a hypercube multicomputer, *IEEE Trans. Parallel Distributed Syst*. (May 1992) 334–349.
[28] V.S. Pandurang and V.S. Balkrishna, *Sampling Theory of Surveys with Applications* (Iowa State University Press, 1970).
[29] D. Schneider and D. DeWitt, A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment, *ACM-SIGMOD*, (1989) 110–121.
[30] M. Stonebraker, The case for shared nothing, *Data Engineering* 9 (1) (March 1986).
[31] J. Ullman, *Database and Knowledge-Base Systems*, Vols. I and II (Computer Science Press, 1988).
[32] J. Wolf, P. Yu, J. Turek and D. Dias, A parallel hash join algorithm for managing data skew, *IEEE Trans. Parallel and Distributed Syst*. (Dec. 1993) 1355–1371.